ACCESSING EXTRA DATABASE INFORMATION: CONCURRENCY CONTROL AND CORRECTNESS

N. GEHANI, K. RAMAMRITHAM, and O. SHMUELI

CMPSCI Technical Report 93-81

Accessing Extra Database Information: Concurrency Control and Correctness

Narain Gehani
AT&T Bell Laboratories
nhg@research.att.com

Krithi Ramamritham*
University of Massachusetts
krithi@nirvan.cs.umass.edu

Oded Shmueli†
AT&T Bell Laboratories
oshmu@cs.technion.ac.il

August 1993

Abstract

Traditional concurrency control theory views transactions in terms of read and write operations on database items. Thus, the effects of accessing non-database entities, such as the system clock or the log, on a transaction's behavior are not explicitly considered. In this paper, we are motivated by a desire to include accesses to such *extra-data* items within the purview of transaction and database correctness. We provide a formal treatment of concurrency control when transactions are allowed access to extra data by discussing the inter-transaction dependencies that are induced when transactions access extra data. We also develop a spectrum of correctness criteria that apply when such transactions are considered. Furthermore, we show that allowing databases to view data which has been traditionally kept hidden from users increases the database functionality and in many cases can lead to improved performance.

*partially supported by the National Science Foundation under grant IRI-9109210.
†Author's current affiliation: Technion - Israel Institute of Technology.

Contents

1 Introduction

Traditional concurrency control theory views transactions in terms of reads and writes on database items. Our goal in this paper is to examine the concurrency and correctness issues that arise when we take into consideration all the data that is used or can be used by transactions, not just what is in the database. Roughly speaking, "extra data" means useful information that is not part of the enterprise's database schema, but is nevertheless useful and may affect what ultimately appears in the "proper database". Such data lives in the shaded zone between the database and the application software; it is our goal to shed some light on its use and ensewing implications.

Thus, for example, we are concerned with data that has always been accessed by transactions but not included traditionally in serializability theory, e.g., the system clock, communication channels, and scratch-pads. We also examine information that can be useful for transactions, such as those that are maintained by lock and recovery managers, e.g., the log and information about transactions waiting for locks (some systems already allow log access for tuning and control purposes). Allowing databases to view such data, namely "extra data", can increase database functionality and can lead to improved performance. For example, users can pose queries that could not be asked before. Also, with access to data such as the predicted behavior of other transactions (e.g., their expected execution times or data access patterns) and transaction management information (e.g., the number of transactions waiting to perform operations on a particular object) transactions can be designed for improved performance.

We elaborate on the benefits of accessing extra data by (1) showing that there are in fact a number of instances where extra data can be viewed and manipulated as first-class data items; (2) examining the properties of such extra data from the viewpoint of the transactions and their correctness; and (3) providing a formal treatment of concurrency control when transactions are allowed access to extra data.

We use the term *database* with its traditional meaning and use the term *extended database* to refer to the database plus the extra-data objects. We introduce an important new concept, that of extra-data independent transactions. Such transactions, intuitively, perform correct database state transitions (in the conventional sense) despite their extra-data accesses. We treat two main correctness requirements: (1) serializability over the extended database and (2) serializability only over the database. We analyze four combinations by considering extra-data independent transactions and extra-data dependent transactions in conjunction with these two correctness requirements. For each such combination we illustrate its usefulness and flexibility in achieving user's goals. This analysis sheds light on the role, usefulness and pitfalls in using extra data.

In many databases, in accordance with protection and security considerations, transac-

tions are allowed access to data on a need-to-know basis. Clearly, when transactions are allowed access to information that is usually within the purview of the transaction management system the protection and security ramifications of such accesses must also be examined. This, however, is outside the scope of this paper.

The rest of the paper is organized as follows. Several examples of using extra data and a discussion of their characteristics can be found in Section 2. A brief introduction to the formalism used to describe the properties of extra data and discuss the correctness of transactions using extra data is given in Section 3. The different correctness notions are discussed in Section 4. Section 5 provides details of these correctness notions and illustrates them via concrete examples. Section 6 contains a discussion of relaxed correctness notions applicable to extra data. Section 7 summarizes the paper and discusses the outstanding issues.

2 Extra Data: Characteristics and Examples

In this section we first give an informal definition of extra data and discuss the characteristics of extra data. Motivation for letting transactions access extra data is also provided via several examples which show that potential for improvement in functionality exists when transactions are allowed access to extra data.

2.1 Characteristics of Extra data

Extra data can be defined as data that is typically not considered to be part of the database. Extra data can be classified into four categories:

1. Data values modified by some entity outside the database system. The system clock is a prime example of this.

2. Data values modified by the transaction management system in response to some request initiated by a transaction. Information about waiting transactions, concurrency control information such as the serialization graph, and the log are examples here.

3. Data values pertaining to (and modified by) the transactions themselves. Estimates of a transaction's (remaining) execution time and data requirements are examples.

4. Data private to a specific set of transactions. A scratch pad used by a set of cooperating transactions to coordinate their activities is an example of this case.

Figure 1 gives a pictorial characterization of the components of an extended database. The shaded area denotes the extra data. Note that not all data in the first two categories may be

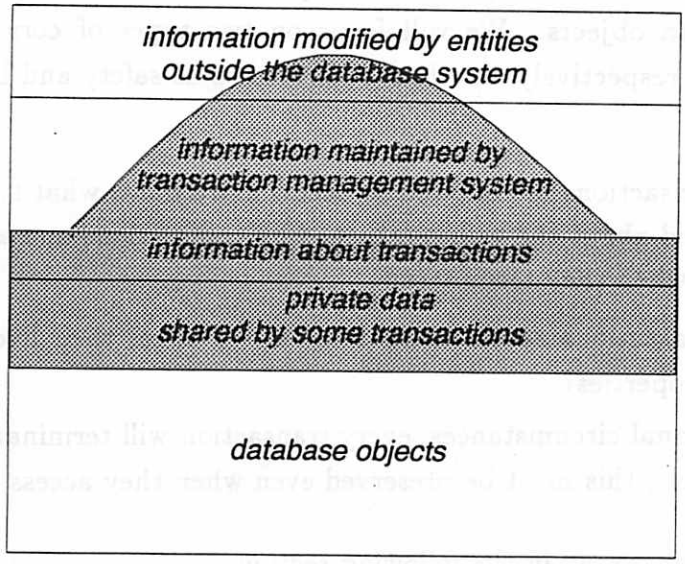

Figure 1: The Extended Database

considered as extra data since only those that can be accessed by transactions are considered to be extra data. It is important to note that unlike database items, extra-data items may not be persistent and may not be *exclusively* accessed (read and updated) by user transactions.

Clearly, updates to a particular type of extra data will be restricted based on the type. For instance, whereas no transaction can update "system updated" or "transaction-management updated" extra data, a transaction might be allowed to update extra data pertaining to itself and not others. These observations have certain important implications for concurrency control. For instance, whereas it may be possible to delay the writing of extra-data items belonging to categories (2) and (3) for concurrency control reasons, this is not possible for extra-data items in category (1). Such delays must be carefully evaluated for their effects on other transactions. For instance, if transactions' commitment or abortion could be delayed (for correctness reasons) purely because of their access to extra data, performance can degrade. Thus, it is important to weigh the pros and cons of transactions' access to extra data before their use is permitted in general. Our goal in this paper however is to understand the concurrency control and correctness issues related to such accesses to extra data since in reality transactions *do* access data outside the database.

Given a database system that allows transactions to access extra data, the following additional questions become relevant:

- What are the operations defined on extra-data objects?
- What are the semantics of these operations? In particular, how do the semantics of the operations defined on an object affect the transactions that invoke these operations?

- What are the correctness requirements imposed on the transactions when they access these extra-data objects. We will focus on two types of correctness-related issues corresponding, respectively, to what is referred to as safety and liveness in concurrent systems:
  - When transactions are allowed to access extra data, what type of guarantees can be provided about the values returned by transactions and about the state of the database when they terminate?
  - When transactions access extra data, will it affect their liveness, namely, termination, properties?

    Under normal circumstances, every transaction will terminate, i.e., abort or commit. Clearly, this must be preserved even when they access extra data.

These questions are answered in the following sections.

2.2 Examples of extra data

- *The System Log.* Traditional database systems hide recovery data from the user. Eliminating this restriction, that is, storing such data as just another object that can be accessed by any user has many benefits [11, 4]. Users can pose queries on the log that cannot be specified in traditional database systems and queries that were not envisioned by the system designers. For example, (1) Find all users issuing transactions that changed item x between Jan 1, 1993 and Jan 31, 1993. (2) What types of transactions are most likely to run on the last business day of the month?
- *Predicted Transaction behavior.*
  - *Knowledge of the set of possible values a transaction will write to a data item.* If a transaction knows the set of possible values it will write to a data item, then it can *proclaim* this to other transactions, which may be in a position to proceed without waiting for this transaction to relinquish its lock [6].
  - *Estimate of execution time and data requirements of transactions.* These will be useful for dealing with transactions in real-time database systems [9]. For example, if it is possible for a transaction to realize that enough time is not available to execute a transaction, or that a feasible schedule is not possible, it could instead invoke an alternative, with a lower computational requirement.
- *Concurrency status of data items.* Knowing how many transactions are waiting to access a data item will allow a transaction to consider alternatives that are likely to reduce its response time.

- *Information about waiting transactions.* In conjunction with the previous item, access to information about waiting transactions can avoid/prevent deadlocks. For example, a transaction can check before it performs an operation on a data item whether or not the operation can proceed without delay. If not, and if the transaction is willing to wait, it will check for possible deadlocks among waiting transactions and if a deadlock is possible, the transaction can take alternative actions.

- *The system clock.* Many transactions possess functionality that depends on the time at which they execute. Examples occur in business applications and in real-time systems.

- *Limited information sharing.* Consider two transactions, executing on behalf of two designers who cooperate while making changes to a design object. They maintain a scratch-pad in which they keep notes about changes made by each. A designer makes an intended change only if the other has not already made it. This scratch pad is accessible only by these designers and is not part of the database, i.e., it can be seen as an extra-data object.

As the above examples illustrate, both the control flow within a transaction as well as a transaction's data manipulation properties may be modified by the extra-data accesses. What a transaction is allowed to do with the extra data that it reads depends on the correctness requirements imposed on the transactions. For instance, they can use it to optimize their actions – but we may require that the semantics of the transformation performed by the transaction be independent of the resulting behavior. More liberal is the case where transactions are allowed to make changes to the database that depend on the values of the extra data, i.e., the database transformation will depend upon the value of the extra data. Of interest here is the repeatability of the transaction in spite of its access to extra data and the relationship of the execution that accesses extra data and the one that does not. We discuss these issues in detail in subsequent sections.

3 Preliminaries and Definitions

In this section we introduce a simple formalism based on the ACTA transaction framework [2]. We also define some of the terms used in the rest of the paper.

Without loss of generality, we assume an object-oriented database: the database is a collection of objects and objects are accessed via database operations.

A transaction accesses and manipulates objects in the extended database, i.e., objects in the database as well as extra-data objects, by invoking operations specific to individual objects.

3.1 Operations, Events, and Histories

It is assumed that operations are atomic and that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s,p)$ to denote the output produced by operation p, and $state(s,p)$ to denote the state produced after the execution of p.

DEFINITION 3.1: Invocation of an operation on an object is termed an *object event.* The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t (for simplicity of exposition assume that a transaction does not contain multiple p's).

When "what the ob is" is clear, we will simply say p_t. $p_t(val)$ is used to denote op that either takes in val as input or produces val as output.

DEFINITION 3.2: A *history* is a partially ordered set of events invoked by transactions. Thus, object events and transaction management events are both part of the history H. The set of events invoked by a transaction t is a partial order denoting the temporal order in which the related events occur in the history. The transaction's partial order is consistent with the history's partial order.

We write $(\epsilon \in H)$ to indicate that the event ϵ occurs in the history H. $\rightarrow$ denotes precedence ordering in the history H and $\Rightarrow$ denotes logical implication.

3.2 Conflicts between Operations

DEFINITION 3.3: Let $H^{(ob)}$ denote the projection of the history with respect to the operations on ob. [1] Two operations p and q on an object ob *conflict* in a state produced by $H^{(ob)}$, denoted by $conflict(H^{(ob)},p,q)$, if

$$\begin{aligned} (state(H^{(ob)} \circ p,\ q) &\neq state(H^{(ob)} \circ q,\ p)) \quad \vee \\ (return(H^{(ob)},\ q) &\neq (return(H^{(ob)} \circ p,\ q)) \vee \\ (return(H^{(ob)},\ p) &\neq (return(H^{(ob)} \circ q,\ p)) \end{aligned}$$

Two operations that do not conflict are called *compatible.*

[1] $H^{(ob)} = p_1 \circ p_2 \circ ... \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations. Thus, a state s of an object produced by a sequence of operations equals the state produced by applying the history $H^{(ob)}$ corresponding to the sequence of operations on the object's initial state $s_0 (s = state(s_0, H^{(ob)}))$. For brevity, we will use $H^{(ob)}$ to denote the state of an object produced by $H^{(ob)}$, implicitly assuming initial state s_0.

(o denotes functional composition.) Thus, two operations conflict if their effects on the state of an object are not independent of their execution order (first clause) or their return values are not independent of their execution order (second and third clauses). Operations p and q are said to *conflict* on object ob, denoted $conflict(ob, p, q)$ if there exists a history H such that $conflict(H^{(ob)}, p, q)$.

The notion of conflict will be used to define different types of correctness when transactions access data as well as extra data concurrently.

3.3 Serializability

In traditional databases, serializability and, in particular, *conflict serializability*, is the well-accepted criterion for correctness. We first define serializability formally since it forms the basis for the correctness notions discussed here.

Let $\mathcal{C}$ be the conflict (binary) relation on transactions in $\mathcal{T}$, where $\mathcal{T}$ is the set of transactions in the history.

DEFINITION 3.4: $\forall\, t_i, t_j \in \mathcal{T}, t_i \neq t_j$,
$(t_i\ \mathcal{C}\ t_j)$ if $\exists ob\ \exists p, q\ (conflict(ob, p_{t_i}, q_{t_j}) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$.

The $\mathcal{C}$ relation captures the fact that two transactions have invoked conflicting operations on the same object and the order in which they have invoked the conflicting operations. Consequently, the $\mathcal{C}$ relation captures direct conflicts between transactions in a history which affect their serialization order. The fact that a serialization order is acyclic is stated by requiring that there be no cycles in the $\mathcal{C}$ relation. All this is formalized below.

DEFINITION 3.5: H, the history of events relating to transactions in T, is *(conflict) serializable* iff $\forall t \in \mathcal{T}, \neg(t\ \mathcal{C}^*\ t)$ where $\mathcal{C}^*$ is the transitive-closure of $\mathcal{C}$.

Suppose t_j has done a write and then t_i does a read. Then, $(t_j\ \mathcal{C}\ t_i)$. Also, given the semantics of read and write, and if we desire failure atomicity, then if t_j aborts then t_i must also abort. That is, t_j has an abort dependency [2] on t_i. Thus, abort dependencies between transactions may also form due to conflicting operations.

As mentioned earlier, an application may desire correctness properties that are weaker than serializability when an extended database is used. A discussion of such correctnes properties can be found in [8]. Since, in this paper, we will be making use of set-wise serializability, we define it in this section. *Set-wise serializability* has been proposed as a correctness criterion for concurrency control in databases that are partitioned into different sets of objects and consistency constraints are supposed to hold among objects in a given set.

For $k = 1, \ldots, k$, let $D_k \subseteq DB$, where DB is the set of items comprising the database, and let C_k be a binary relation on transactions in $\mathcal{T}$. Let H be the history of events invoked by transactions in $\mathcal{T}$.

DEFINITION 3.6: $\forall k \in \{1 \ \ldots \ n\}, \forall\, t_i, t_j \in \mathcal{T},\ t_i \neq t_j,$
$(t_i\ C_k\ t_j)$ if $\exists ob \in D_k\ \exists p, q\ (\mathit{conflict}(ob, p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$.

Thus, C_k is C applied to a particular D_k.

DEFINITION 3.7: H is *set-wise serializable* if $\forall t \in \mathcal{T}\ , \forall D_k\ , 1 \leq k \leq n\ , \neg(t\ C_k^*\ t)$.

In [10], each D_k is said to be an *atomic data set.* For the purposes of this paper, given an extended database, we can consider (1) the extended database to form a single atomic data set or (2) the database to form one atomic data set and all the extra-data items to form another atomic data set. That is, in (2), a cycle of C relationships formed by accesses to the database as well as by accesses to extra data will be ignored since only C_k^* needs to be acyclic where C_k pertains either to the database or to the extra data.

3.4 Extra-data Independent Transactions

A *transaction* is a program that may access the extended database. The reading and writing of extra data may in principle affect both the operations that transaction T performs while executing as well as its flow of control.

Consider a projection of the history containing only the operations defined on the database items. An execution history involving extra data is *database serializable* if this projection of the history is (conflict) serializable. By focusing on just this projection, the accesses to the extra-data items are viewed as accesses to local variables. Consider a transaction T that may access extra data.

DEFINITION 3.8: A *conventional* transaction is one that does not access extra-data items.

DEFINITION 3.9: Transaction T is said to be *extra-data independent* if there exists a conventional transaction T' satisfying the following condition: for all database serializable histories in which T participates[2], T can be replaced by T' in the serial history so that T' performs the same transformation to the database in the serial history as performed by T.

[2]Recall that we use the term *extended database* when we want to refer to the database plus the extra-data items.

Observe that if T is extra-data independent then, when applied to a database state s, it yields exactly what T' yields when applied to s (here the history consists of a single transaction). If T is extra-data independent then, as far as the serializability of the changes to the database is concerned, we can think of the actual transaction T as a "surrogate" for T'. If an execution involving T is serializable then by substituting T' for T, in a serialization order, we obtain a serial history with the same effect on the database state.

Henceforth, we refer to any T' according to the above definition for T, as *equiv*(T). (In general there may be a number of such T's, for our purposes they are all equivalent.) A transaction T which is not extra-data independent, is called *extra-data dependent.*

Let us consider a simple example to clarify the notion of extra-data independence. Suppose a transaction is invoked to make a single car rental reservation. It chooses among car companies based on the anticipated delay due to waits. The transaction uses extra data as control information to find the path with the shortest waiting time:

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    if (num_waiting_read_write(avis) < num_waiting_read_write(hertz))
        hertz->reserve(...);
    else
        avis->reserve(...);
    ...
}
```

This is an example of a transaction that is *not* extra-data independent: either an Avis car is reserved or a Hertz car is reserved.

Consider a transaction invoked on behalf of another user who intends to reserve an Avis car in Los Angeles and a Hertz car in Houston. The following transaction accomplishes this while attempting to minimize the waiting time.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    if (num_waiting_read_write(avis) > num_waiting_read_write(hertz))
        hertz->reserve(...); avis->reserve(...);
    else
        avis->reserve(...); hertz->reserve(...);
```

```
    ...
}
```

This transaction *is* an extra-data independent transaction that is equivalent to the following transaction that does not access extra data.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    avis->reserve(...); hertz->reserve(...);
    ...
}
```

Consider a database serializable history where all transactions are extra-data independent. In running the original transactions serially, extra data may be different or may not exist, which could lead to a change of control flow and/or values written. But, substituting $equiv(T)$ for each T isolates the important facets of the execution, namely its effect on the database state, from other facets as embodied in extra data.

Given a transaction (program) T, it is undecidable whether an $equiv(T)$ exists. So, we cannot delegate the task of verifying the existence of $equiv(T)$ to an automatic tool. An alternative is to have the user supply $equiv(T)$ and an equivalence proof. This is similar to having the user certify a transaction as doing a "correct" state transformation in the traditional formulation of concurrency control theory.

We now discuss some practical aspects of constructing extra-data independent transactions:

- Divide transactions into two categories: transactions that read and write database objects only, and those that read and write extra data and read database objects. Clearly, such transactions are all extra-data independent.

- As a little more interesting case, categorize as follows: transactions that read and write database objects but are allowed to write extra data, and transactions that read or write the extra data and read database objects. Here again, all the transactions are extra-data independent.

- Allow any transaction to read and write extra data. If values read from extra data affect the final writes of a transaction, then the transaction may not be extra-data independent. Using data flow analysis, we can verify that extra-data information does not propagate into the database. Then, the only effect extra data may have

on a transaction is manifested through flow of control. To make a transaction extra-data independent, construct the transaction so the effect on the database is the same regardless of the flow of control.

4 Correctness Notions when Transactions Access Extra data

It should be obvious by now that different types of correctness notions can be applied in the context of extra data. In this section, we discuss several such correctness notions. The next section illustrates them with concrete examples of extra data.

Recall that we are interested in serializability as the correctness criterion with respect to the data. We may desire serializability w.r.t. accesses (a) to data in the extended database or (b) to data just in the database. By serializability with respect to just the database data we mean set-wise serializability with the only set of interest being the set of data items in the database. Intuitively, it says that if the actual transactions are run serially and whenever an extra-data access is performed in this serial execution the same values as in the actual execution are "magically" supplied, then transactions will produce the same final database state as in the actual execution.

With regard to the transactions, (a) transactions may be required to be extra-data independent or (b) transactions could be extra-data dependent. Thus we are led to four possible combinations of serializability-based correctness requirements:

1. Achieve serializability of accesses to the extended database. Also, the transactions must be extra-data independent.

   Given our definitions in Section 3, this implies that we should consider conflicts over database objects as well as over extra-data objects and ensure that the induced C relation is acyclic. In addition, the effects on the database will be as though the transactions did not access any extra-data object.

   It is important to understand the implications of this correctness property. The fact that transactions are extra-data independent implies that they behave – with respect to changes to the database – as though they did not even access the extra data. Serializability of all transactions, especially those that view just the extra data implies that these observe the extra data as any serial "observer" would.

2. Achieve serializability of accesses to the extended database.

   We need to just ensure that the induced C relations are acyclic.

   The removal of the extra-data independence requirement for transactions implies that the transactions may produce results that reflect the fact that they accessed extra

data. As an example, consider a transaction that updates account balances with the day's deposits and withdrawals. Also, it reads the calendar and adds the interest for the month to the balances only if it happens to be the last day of the month.

3. Achieve serializability of accesses to (just the objects in) the database. Also, the transactions must be extra-data independent.

   This requires the C relations due to conflicting operations on the database objects to be acyclic. That is, the database is one of the two atomic data sets in the system, the other formed by the extra-data objects. Serializability is the correctness criterion applicable to the former. In addition, the effects on the database are required to be as though the transactions did not access any extra-data object.

   One of the implications of this correctness criterion is that an observer of extra data may not produce serializable results since conflicts over accesses to extra data are not considered in determining the acyclicity of C relations. These results should be taken just as "hints". However, all changes to the database are as though the transactions producing the changes never accessed any extra data.

4. Achieve serializability of accesses to the database.

   In contrast with the previous correctness requirement, transactions need not be extra-data independent.

   Here changes to the database may depend on the transactions' accesses to extra data. Note that since the extra-data conflicts are not considered in determining which transactions conflict, transactions may produce database changes that are affected by extra-data accesses. That is, this correctness criterion *does not* guarantee consistency of the database. This is because values written to items in the database may depend on the values of extra data read by transactions.

We now show what are the concurrency implications of the above correctness criteria. Note that the larger the number of histories considered acceptable by a given correctness criterion, the more the potential for improving performance.

Theorem: Let H_i denote the set of histories considered to be correct according to correctness criterion i. Then, $(H_1 \subseteq H_2 \subseteq H_4) \wedge (H_1 \subseteq H_3 \subseteq H_4)$

Proof: $(H_1 \subseteq H_2)$ as any history qualifying under H_1 would qualify under H_2, as H_2 does not require that transactions be extra-data independent. The reasoning is identical for $(H_3 \subseteq H_4)$. $(H_1 \subseteq H_3)$ as both deal with extra-data independent transactions, but H_1 requires serializability to apply to the extended database whereas H_3 only to the database. $(H_2 \subseteq H_4)$ for identical reasons.

H_2 and H_3 are not comparable. We prove this by first giving an example of a history that is in H_3 but not in H_2 and vice versa.

Consider a database containing one data item x and one extra-data item y, both of type integer. Let R_i (W_i) denote a read (write) operation by transaction i. Let transactions t_i, $i = 1, 2$, execute the following transaction text:

$$\begin{array}{ll} l2 = R(y); & \\ y = i; & (W(y)) \\ l1 = R(x); & \\ x = X + 1; & (W(x)) \end{array}$$

$l1$ and $l2$ are local variables. Consider the following history:

$$H = R_1(y); R_2(y); W_2(y); W_1(y); R_1(x); W_1(x); R_2(x); W_2(x);$$

Here, H is in H_3 but not in H_2. This is because H is serializable over the database but not over the extended database. Also, both transactions are extra-data independent, they increment x regardless of the value of y.

Let us change t_i, $i = 1, 2$, to be:

$$\begin{array}{ll} l2 = R(y); & \\ l1 = R(x); & \\ x = Y; & (W(x)) \\ y = i; & (W(y)) \end{array}$$

Consider the following history:

$$H' = R_1(y); R_1(x); W_1(x); W_1(y); R_2(y); R_2(x); W_2(x); W_2(y);$$

Here H' is in H_2 but not in H_3. This is because H' is serializable over the extended database (it is actually serial) and transactions are extra-data dependent. *End Proof*

5 Concrete Examples of Extra Data with varying Correctness Requirements

In this section we discuss in detail each of the correctness notions introduced in the last section, providing concrete examples of extra data for which these correctness notions are applicable.

5.1 Serializable Accesses to the Extended Database– with Extra-Data Independent Transactions

Here, we require transactions to be extra-data independent. To ensure serializability over accesses to the objects in the extended database, we must consider the conflicts resulting from accesses to both the data and the extra data.

Consider a transaction t' which performs operations op_1, op_2, and op_3, in sequence, on each data item in a set D. We now show how t' can be implemented as a transaction t which uses extra-data items in such a way that it has the potential for better performance in a parallel environment. That is, t' will be $equiv(t)$.

t consists of three (sub)transactions t_1, t_2, and t_3. t_1 performs op_1, t_2 performs op_2 and t_3 performs op_3. t_1, t_2, and t_3 operate in a pipelined fashion, i.e., after t_1 does op_1 on a data item d, t_2 does op_2 on d and then t_3 does op_3 on d. Proper control of the transactions' actions is achieved via work queues q_1 and q_2 which are extra-data items. t_1 inserts the id of the data item on which it just completed op_1 into q_1. t_2 deletes the id in the front of queue q_1 and after doing op_2, it inserts the id into q_2. t_3 removes an id from q_2 and performs op_3 on the corresponding data item. When q_1 and q_2 are empty and t_1, t_2, and t_3 have completed their ongoing operations, they all commit together. If any of them aborts, all abort.

Transaction t, consisting of t_1, t_2, and t_3, has the same effect as t', with respect to operations on D. That is, t is extra-data independent and t' is $equiv(t)$. Also, the following axiom suffices to show that interactions over the extra-data items q_1 and q_2 are also serializable where the serialization order is t_1, t_2, t_3.

> $(t_i \; C \; t_j))$ if:
> $\exists k(insert_{t_i}(k) \rightarrow remove_{t_j}(k))$.
>
> This states that t_j must remove id k only after it is inserted by some t_i and (hence) t_i should precede t_j in the serialization order.

What this example shows is that it is possible to realize a given transaction in such a way that even though the implementation uses extra-data items, its behavior with respect to the rest of the transactions and the effect on the database will be the same as the original transaction. The motivation here is to rewrite the transactions in a way that allows us to execute components of the transactions in parallel thereby improving the performance of the system. The queues allow the transaction components to synchronize their activities so as to achieve the desired functionality.

Another example of extra-data independent transactions producing serializable access to the extended database occurs in the multi-database concurrency control scheme proposed in [3]. Here, to ensure the serializability of transactions that access multiple (autonomous) database sites, the following scheme is used. Every site has a special "ticket" that all global transactions that visit the database at that site are expected to read and write. In this case, the ticket is the extra-data item. The global database serialization order that results is the same as the order in which this ticket is visited by the transactions and so serializability is achieved over the extended database. Also, since what each transaction does is not affected by the value of the ticket, the transactions are extra-data independent.

5.2 Serializable Accesses to the Extended Database

Here we remove the extra-data independence requirement imposed on transactions. For concreteness, we consider the database log as an example of extra data which can be accessed by transactions in the course of their execution [4]. Both committing and aborting transactions write log records. Transactions can also read the log to perform queries. We require serializability of all the data items in the extended database, in this case, the database plus the log.

Let us assume that the commitment or abortion [3] of a transaction results in the writing of a single log item containing all the relevant information about the transaction. Conceptually, the log can be considered as a linear object that grows in one direction. Each item in the log has an id (its LSN; i.e. log sequence number).

$append_{t_i}(k)$ denotes the appending of a log item pertaining to transaction t_i; when the operation completes, the id of the appended log item is in k. The value of k is one larger than the id of the previously appended log item.

$read_{t_i}(k)$ denotes the read operation on log item k by transaction t_i; this item should already exist in the log for the read to be successful. Otherwise, the read fails.

$last(k)$ can be used to determine the id of the last item in the log. This id will be in k when *last* completes.

Given this, the following axioms state the properties of these operations in terms of their effects on the resulting C relationships. Specifically, $\forall\ t_i,t_j\ t_i \neq t_j\ \ (t_i\ C\ \ t_j)$ if:

1. $\exists\ k,k'\ (append_{t_i}(k) \rightarrow append_{t_j}(k'))$

   Since both aborting transactions and committing transactions write to the log, transactions append to the log in the same order in which they commit or abort. Also, every transaction, once it begins execution, will commit or abort. Thus, when a transaction t_i appends an entry into the log, a transaction t_j that has not yet committed or aborted, i.e., t_j is a transaction in progress, will write its log entry after t_i's entry; Since t_j updates the queue after t_i updates it, $(t_i\ C\ \ t_j)$, that is, t_j must appear after t_i in the serialization order.

2. $\exists\ k\ (append_{t_i}(k) \rightarrow read_{t_j}(k))$

   This states that if t_j reads log record k, the read should occur after the write of record k by some t_i.

3. $\exists\ k\ (append_{t_i}(k) \rightarrow last_{t_j}(k))$

[3] Transactions can abort for one of many reasons, including unilateral aborts.

This states that if t_j, via the *last* operation, "knows" that the last log record to be written was the k^{th} record (written by transaction t_i) then transaction t_i should precede t_j in the serialization order.

4. $\exists k, k'\ last_{t_i}(k)) \rightarrow (append_{t_j}(k')$

   A transaction t_j that has not yet committed or aborted when a transaction t_i performs the *last* operation will write its log entry after t_i executes *last*. That is, t_i observes the queue before t_j updates it and hence t_j will have to follow t_i in the $\mathcal{C}$ ordering.

The four conditions define the $\mathcal{C}$ relationships, i.e., the serialization ordering requirements induced by transactions' read, last, and append operations on the log. For each operation, the axioms consider the order of occurrence of the other operations that produce serialization ordering requirements.

Given that $\mathcal{C}$ relationships result not only from the invocation of operations by transactions on database objects but also from the invocation of operations on the log objects, the system should make sure that the acyclicity of the $\mathcal{C}^*$ relation is an invariant. This is the safety-related correctness property that must be satisfied usually, and is still the case when accesses to the log are considered. The practical implication of the above axioms on transaction management is the following: When an operation is performed on the log, the system must note the $\mathcal{C}$ relationships induced by the operation, in light of the above axioms, and must ensure that the $\mathcal{C}$ relation is acyclic.

Here are additional correctness requirements imposed on read and last operations:

- $read_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j\ (append_{t_i}(k) \rightarrow read_{t_j}(k)))$

  This states that t_j reads log record k after the write of record k by some t_i.

- $last_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j\ (append_{t_i}(k) \rightarrow last_{t_j}(k))$.

  This states that if the *last* operation executed by t_j returns k, then the k^{th} record should have been written by some transaction t_i.

Traditionally, cyclic $\mathcal{C}$ relations can be handled through transaction abortions. But with transactions being able to access the log, we must now worry about the liveness of the transactions because aborting and committing transactions that access the log form $\mathcal{C}$ relationships which must not create cycles. The following theorem allays any fears in this regard.

We assume here that the changes done by a transaction can be undone upon its abort without forming further $\mathcal{C}$ relationships. This is a valid assumption if transactions perform read and write operations on data [1].

Theorem: Given a set of existing C relationships, it is always possible to force transactions' operations including those on the log to occur in an order such that C^* will be acyclic.

Proof: We will show that the theorem holds by considering each of the three operations.

Let us consider a transaction t_i that desires to terminate (commit/abort), i.e., append to the log. If no cycles exist in the current set of C relationships and t_i's termination will not cause a cycle due to the additional C relationships caused by the axioms, then t_i can terminate. Otherwise, there is a topological sort of the transactions which conforms to the current C relation and there is a transaction t_1 which has no predecessor in this sort. Since transaction t_i desires to terminate, we should make sure that all its predecessors in the sort have also terminated. In turn, these predecessors may have predecessors that need be terminated, and so on recursively. This recursion stops with t_1. We can simply run t_1 until completion (if no t_k C t_1, $k > 1$, relationship will be formed – it may be formed only due to data contention) or abort t_1. Then, we can treat t_2 in the same way, and so on. So, we know that there is a way to achieve the termination of some arbitrary t_i.

Now suppose t_i desires to perform a *read* operation on the log. If the read is successful, according to axiom 2, it should be serialized after the transaction, t_j, that wrote the log item. Since t_j has already terminated (and that is why this record is in the log) this serialization ordering requirement is automatically satisfied. Suppose the read is not successful and so t_i has to abort. From what we said in the previous paragraph, this abortion will be successful.

Finally, suppose that t_i desires to perform a *last* operation. Suppose *last* returns k. The transaction t_j that wrote the k^{th} log record must be serialized before t_i, as per axiom 3. Since t_j has already committed this is automatically satisfied. *End proof.*

We will show now that, in practice, it might be better to "look ahead" at (known) future C relationships to reduce the number of abortions caused by cycles in the C relation. Consider the following scenario. Suppose (t_j C t_i) already exists. (For example, t_j does a *write* of some object in the database and then t_i does a *read*. If t_j aborts (for some reason) then since we require failure atomicity, t_i must also abort. (See Section 3.)) Suppose t_i performs the *last* operation and t_j is yet to terminate. That is, t_j will perform an *append* operation later, and hence, by axiom 4 will produce (t_i C t_j). Thus, if we wait until t_j's append to notice the cycle, the only way to break the cycle then is to abort t_i. But instead, given axiom 4, if we had allowed t_i to perform *last* only after t_j terminates, then we could have perhaps committed both t_i and t_j. Thus, in general, ensuring that there are no C cycles can be achieved by delaying operations whenever possible.

Also, as the following example shows, in some situations, forcing the abortion of predecessors in the C relation (to allow a transaction t_i to abort) may remove the need to abort t_i. Assume that t_i, t_j, and t_k are the only transactions in the system and the following are true:

$(t_i \; C \; t_j)$ and $(t_j \; C \; t_k)$. Suppose t_i and t_k are involved in a deadlock. If t_i is aborted, things will be fine. The abortion of t_k however will produce $(t_k \;\; C \;\; t_i)$ – since then t_k's abortion (and hence the appending of its log record) will precede that of t_i – and hence cycles in the C relation. Let us see how the "algorithm" suggested by the proof of the above theorem would handle this. The topological sort consistent with the C ordering is (t_i, t_j, t_k). So when t_k is aborted, it will first force the abortion of t_i and t_j. Once t_i is aborted, the need to abort t_k disappears and with that the need to abort t_j as well. This implies that the deadlock resolution algorithm must reevaluate the need to abort a transaction in order to avoid unnecessary aborts.

Note that in the case of the log, as long as the history resulting from concurrent transaction executions is serializable, correctness is considered to be preserved. The log is considered like any other data item and is considered as being part of the database.

When other extra-data objects are accessed by transactions and serializability remains the correctness criterion, the semantics of these other objects should be specified just as we dealt with the log and then we must show that safety and liveness properties of transactions are kept in tact.

It is important to realize that when transactions are allowed access to the log and serializability remains the correctness criterion, it may delay the commitment or abortion of other transactions. Specifically, because of axiom 4, once a transaction t_i performs *last*, no other transaction can commit or abort until t_i commits. Such a delay can affect performance. However, if one were to relax the correctness criterion, say by requiring something akin to the lower degrees of isolation of traditional databases, then this negative impact can be reduced or eliminated. Specifically, one could opt for non-repeatable reads of the log. We return to this issue in Section 6.

5.3 Serializable Accesses to the Database – with Extra-Data Independent Transactions

Consider the following example in which 2PL is used for concurrency control. Suppose a transaction t' holds a write lock on a data item and another transaction t desires to read that data item. Under 2PL, t would be made to wait. However, suppose t' gives t (or any other transaction) an indication of what the possible values it might write are, then t might be able to proceed with its computations using this information. This is the idea underlying proclamations [6]. t' proclaims the set of possible values that may be written so that transactions such as t may be able to proceed without waiting. In this section we show how proclamations can be viewed in terms of transactions interacting via extra-data items.

A transaction using proclamations is implementing an ordinary (sequential) program text, the sequential program is called the *underlying code*. At any point in time such a

transaction maintains a number of computation threads of the underlying code, each based on a different set of values for (read) database items, obtained through proclamations of other transactions. Still, the collection of all these threads is internal to a single transaction as far as the system is concerned, hence, in particular, threads cannot interact among themselves to create a deadlock situation. Each thread may be ready to proclaim possible future values for some item X, once all threads are ready to proclaim for X, the union of their individual proclamations is proclaimed in an extra data item x associated with X. Observe that proclamation is a voluntary operation, a transaction is not forced to proclaim.

In what follows assume that 2PL is used and that every transaction reads a data item before writing to it. The "right" to proclaim for X is awarded only to the transaction that holds a write lock on X. Suppose a transaction proclaims a set of values for X in x. *This set must contain the value the transaction read for* X. The transaction may subsequently write to x subsets of this set, which are monotonically decreasing (i.e. each new set of values is a subset of the preceding one). Before a transaction commits it writes single values to each of the variables x in which it proclaimed, that value must appear in x. That value is also the value that it writes to the database item X. Before a commit decision is reached, no writes are performed into the database.

A transaction t may read proclamations for some items as follows. When a read request on X is blocked on a lock, a proclamation read is performed. If the variable x is non-empty, the set of values in x is read by the transaction t. (Otherwise, the transaction waits until either variable x is non-empty or its read lock is granted.) The values read this way are produced by another transactions t'. As stated, each set proclaimed by t' for X must contain the value t' read for X. After viewing the proclaimed values, t may either

1. Proceed, executing conceptually in parallel, one instance of its underlying code for each combination of proclaimed values for items it has read up to now. t keeps maintaining its read request on X.

2. Wait for a read lock on X.

3. Wait for a refinement of the proclaimed values.

In the first case, transaction t can commit if for each of the parallel computations, for each item, it intends to do the same final write operations. (For example, consider a transaction that depends on whether $X < 100$ or $X \geq 100$ and not on a particular value of X. If X is proclaimed to be updated (in the future) from 60 to 62, 65, or 67, all parallel computations will produce the same result.) If this is not the case, the transaction may be aborted or refinement may be done in one of the following ways:

1. Do another read on x. Since the value(s) read are a subset of what was read before, this

may be sufficient to resolve the problem by "killing" some, by now irrelevant, threads of computation.

2. Pick a transaction t' from which t reads a proclamation which needs be refined and make a dependency ($t \ C \ t'$), provided it keeps C acyclic. The "before value", i.e., the value t' reads before writing to it, may be used (Recall that this before value is part of the proclaimed set and is marked as such).

3. Another possibility is to wait until some values are resolved in some proclamations that t read. Until then the transaction simply waits.

4. Commits while writing a ***conditional multi value*** which may be used by other transactions, and will be refined "behind the scenes" by the system as more information becomes available. In this case the read locks held by the transaction, on items whose value (i.e. set) needs be refined, are held until no more refinement is needed. (This option contradicts the requirement of writing a single value for each relevant item X at commit time; we shall not discuss it further in this paper.)

If a transaction aborts, its proclaimed values, if any, are erased prior to its abort. If transaction t read a proclamation, say in x for X, from t' and then t' aborts, this does not affect t directly. If t needs refinements of the value of X it will be told to use the value that t' has read for X (i.e., the before-value) which is so marked within the set proclaimed for X in x.

We are now in a position to give a formal definition of the operations connected with proclamations and also state the C relationships that form when these operations execute. Let $p\_write_{t_i}(x,p)$ denote t_i's proclamation of the fact that it intends to write one of the values in the set p. It does so by placing p in x associated with data item X. $p\_read_{t_i}(x,p)$ denotes the reading by t_i of the set p from x. t_i and t_j denote two different transactions. Let $write_{t_i}(X,V)$ denote t_i's actual write into database item X of value V and $read_{t_i}(X,V)$ denote t_i's actual read from database item X of value V. $\forall\ t_i,\ t_j,\ i \neq i$ define ($t_i\ C\ t_j$), if any of the following six situations hold:

1. $(write_{t_i}(X,V) \rightarrow p\_write_{t_j}(x,p))$

2. $(write_{t_i}(X,V) \rightarrow write_{t_j}(X,W))$

3. $(write_{t_i}(X,V) \rightarrow p\_read_{t_j}(x,p))$

4. $(write_{t_i}(X,V) \rightarrow read_{t_j}(X,W))$

These four axioms states that if transaction t_i has performed a "true" database write (i.e., a *write*), then any subsequent operation by a transaction operating on either X or x forces ($t_i\ C\ t_j$), $i \neq j$.

5. $(p\_write_{t_i}(x,p) \rightarrow p\_write_{t_j}(x,p'))$

   This states that if two different transactions make proclamations with respect to a data item X, their serializability order is the same as the order in which proclamations occurred.

6. $(p\_write_{t_j}(x,p) \rightarrow p\_read + resolve_{t_i}(x,p_b))$

   This relates a write to x and the reading of x. Recall that p contains not only the values a transaction might write but also the value of X it read before any write took place. We denote this before value by p_b.

   read+resolve is an operation denoting a decision to resolve using a before value; it only interacts with the proclaiming transaction; it must also be the case that this operation is invoked only if the proclamation had been read previously, see below. This states that if t_i reads the before value proclaimed by t_j, then it should precede t_j in the $\mathcal{C}$ ordering. This happens if t_i cannot wait for the refinement to occur and so commits based on the before value.

No $\mathcal{C}$ relationship is imposed if t_j reads the proclaimed set contained in x after t_i writes a proclamation to x. This is because, as we saw earlier, if t_j's behavior is invariant to the values that t_i may write to x, it can be placed either before or after t_i in the $\mathcal{C}$ ordering. Also, given the motivation underlying proclamations, we do not require repeatable reads. In fact, we depend on proclamations producing monotonically refined values. Thus, any conflicts due to a *p_read* operation preceding a *p_write* operation are not considered. In this sense, a correctness criterion weaker than serializability, specifically, one implied by degree-1 isolation (see Section 6), is all we need for the extra-data x.

We also require the following two correctness requirements to hold:

- If $(p\_write_{t_i}(x,p) \rightarrow p\_write_{t_i}(x,p'))$ then $(p' \subseteq p)$.

  This states that proclamations for X by t_i should be monotonically decreasing.

- If $p\_read + resolve_{t_j}(x,p_b))$ then $(p\_read_{t_j}(x,p) \rightarrow p\_read + resolve_{t_j}(x,p_b))$

  That is, a transaction must have read the proclamation it resolves on.

Observe that the histories produced by using proclamation are not repeatable if transactions are run in the finally determined serial order. This is because transactions will view different values in extra-data items. However, if instead of the actual transaction t we plug in $equiv(t)$ in the schedule in which the transactions are run serially will be the same as in the actual execution. This intuition is made precise by the theorem below.

Theorem: Proclaiming transactions $t_1, \ldots, t_n$ produce a database state that can be produced by some serial execution of transactions $t'_1, \ldots, t'_n$ where t'_i is the transaction executing the underlying code of the proclaiming transaction t_i [4].

Thus, t'_i is $equiv(t_i)$. Also, all committed transactions are extra-data independent provided we only consider histories produced by proclaiming transactions.

Proof: Consider the history produced by an execution which contains both the database and the extra-data accesses. The C relationship is acyclic. So, we can list the transactions in a linear order $t_1 \ldots t_n$ consistent with C. Consider the hypothetical computation $t'_1 \ldots t'_n$, where if t_i has aborted so does t'_i.

We prove by induction that for all k, if during its actual execution t_k writes a value into a data item x, t'_k writes the same value in the hypothetical execution and hence t'_k is $equiv(t_k)$.

Basis ($k = 1$): If t_1 aborted then it never wrote into the database and the claim is true as t'_1 is aborted as well. In case t_1 did not abort, in the actual execution, all reads done by t_1 are either from the initial database state or from proclamations done by transactions that have not yet committed. Otherwise, there would have been a committed transaction prior to t_1 that wrote into an item X that t_1 tried to read and this would have induced a C relationship. But given that t_1 is the first transaction in the linear order that is consistent with C, this cannot be the case. All the parallel computation threads within t_1, in particular including the computation that considered the combination of data values in the initial database, produce the same answer. But, this particular computation is t'_1.

Induction ($k > 1$): Again, if t_k has aborted, we are done. Otherwise, inductively, the claim holds for $t_1 \ldots t_{k-1}$. Consider t_k and t'_k. Any value read by t_k was either obtained via a direct database read or was a proclamation that contained a value (the "before value") that was obtained by a direct database read. All the real database values must have been produced by $t_1 \ldots t_{k-1}$ or taken from the database because of the conditions involving *write*. Also, these are the same values that t'_k reads, again because of the axioms involving *write*. As argued in the basis case, since all of the parallel computations within t_k agree on the values they produce, the particular computation which works with the combination that corresponds to the "before values" also agrees. But this is the t'_k in the hypothetical serial execution. *End Proof*

5.4 Serializable Accesses to the Database

As was mentioned in Section 4, here we consider the database to form one atomic data set and the extra-data items to form another atomic data set. Serializability is the requirement for the former. This may be useful for transactions with particular correctness requirements.

[4] That is, the code of t'_i is run in parallel by t_i on all combinations of proclaimed values.

Consider a modified version of the first car rental transaction from Section 3.4. Here, the available credit is updated where the amount depends on the car rental company chosen.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    if (num_waiting_read_write(avis) < num_waiting_read_write(hertz))
        hertz->reserve(...);
        master_card->credit_reserve(100);
    else
        avis->reserve(...);
        master_card->credit_reserve(75);
    ...
}
```

This transaction is clearly not extra-data independent since the specific data items it accesses depends on the values of the extra data read by it. Also, the credit reservation amount, 75 or 100, depends on the extra data as well. However, the application writer may not care from which company the car is rented as long as exactly one car is rented. Thus, if all the transactions have such a behavior, we just need to ensure that conflicts due to accesses to the database are serializable. This ensures that the database is always in a consistent state as far as applications are concerned.

For another example, consider two transactions t_i and t_j that access disjoint parts of the database but what one transaction accesses is dependent on what the other accesses. The transactions communicate via communication channels that can be modeled as extra-data items which are read and written by the transactions. That is, t_i writes into the channel the ids of the data items it has accessed and t_j reads these, and vice versa. Here, we have extra-data dependent transactions that produce serializable behavior with respect to the database accesses. Since t_i and t_j access different parts of the database, they don't even conflict with each other. Here again, we have extra-data dependent transactions that produce serializable behavior with respect to the database accesses. There is a subtle difference here from serializability in that if we look at the resulting serial schedule and re-run the programs corresponding to t_i and t_j we will not necessarily get the same overall state changes because of accesses to the extra data and the extra-data dependence of t_i and t_j. However, if we rerun the t_i and t_j so that they reflect the exact database accesses they made when they originally ran then the rerun will produce the same database state as the original schedule.

6 Other (Application-Specific) Correctness Notions

Thus far, we have assumed that we would like to extend serializability as the correctness criterion for transactions accessing extra data. But, as we alluded to at several places, it might be appropriate to relax serializability, as has been suggested even for transactions accessing just the database. The added motivation for this in the context of extra-data access is that access to extra-data items, such as the log, which lie in the processing path of every transaction must be allowed with minimal or no impact on performance. Since relaxing correctness requirements is one way to achieve this, serious consideration must be given to it.

Let us now consider some weakened isolation requirements [5] that have been suggested and adopted in practice for transactions accessing the database. In the context of read/write objects, degree-2 isolation ignores conflicts resulting from a read followed by a write. Such a requirement leads to lack of repeatable reads. Degree-1 isolation ignores, in addition, conflicts resulting from a write followed by a read. This permits the read of an object, writes on which have not yet committed, without forming a C relationship between the writing and the reading transaction. Degree-0 isolation ignores all dependencies. Let us consider some examples applying these ideas to extra data.

- Suppose transaction t_i accesses the system maintained current time. A subsequent update of the current time by the system clock will not affect t_i if degree-2 isolation is in effect.

- Suppose the transaction management system updates the wait-for-graph on behalf of a transaction that waits for a lock on an object. Another transaction, which desires to know the length of time it will be forced to wait under current circumstances, views the wait-for-graph. Under degree-1 isolation, it will be allowed to proceed without forming any additional C relationships.

In both these cases, the extra-data items of interest are not directly updated by the transactions. If such transactions do not require the *repeatable read* property as guaranteed by standard concurrency control mechanisms such as locking, then they can afford to allow update operations to take place following their reads without incurring serialization ordering requirements.

With the above in mind, if we consider the log as extra data, the removing of axiom 4 of the log semantics has the effect of foregoing the repeatable read requirement since two invocations of *last* may now identify two different log records as the last record in the log. Eliminating axiom 4 eliminates the performance penalty mentioned at the end of Section 5.2 while still providing a well-understood form of correctness.

Now let us consider extra data that can be written by transactions. Suppose transactions t_1 and t_2 cooperate by reading from and writing into a data structure that the two of them share. Thus, t_1 can read it after t_2 writes it and t_2 can again write into it, all without forming any C relationships. Such a data structure can be modeled as extra data. If t_1 reads and t_2 writes, we get degree-1 isolation. If both write and read, degree-0 isolation results.

To minimize the impact on the transaction management system, it might be useful to design special extra-data access functions in the case of extra data that can only be read by transactions. Similarly, for extra data used to achieve cooperation or coordination, canonical forms of the same could be defined at the user level and system support provided to have minimal impact on performance.

To summarize, application specific correctness requirements applicable to extra data can be specified in terms of (a) conflicts that must be ignored in determining C cycles or (b) conflicts that induce C relationships. Let us consider an example of the latter to show its general applicability. Suppose we want transactions to possess the temporal causality property which requires that if two transactions read some extra data in a certain order with an intervening write to the extra data, their serialization should reflect the order of the reads.

$$(read_{t_i} \rightarrow write_{t_j}) \wedge (write_{t_j} \rightarrow read_{t_k}) \Rightarrow (t_i \; C \; t_k)$$

A concrete example of such an extra-data object is the system clock. What the above requirement demands is that a transaction (t_k) that executes later (according to clock time) than another transaction (t_i) must also be serialized after the other transaction.

7 Discussion

Recently, there have been many extensions to the classical work on concurrency control. One extends and elaborates the structure of data items, viewing them as abstract data type objects, thus exploiting the semantics of the operations for better concurrency control. Another relaxes the serializability correctness criterion by imposing instead specific constraints on acceptable schedules [7]. The work reported herein bears resemblance to these extensions in that, technically, we also view extra data as objects with arbitrary operations defined on them and impose some restrictions on acceptable schedules. However, these are by-products of our main interest, that of enlarging the set of transaction accessible data to include structures that are traditionally either (a) hidden within, and are internal to, the database system itself or (b) local to a set of transactions. We have examined the consequences of such accesses and in doing so we are forced to use extensions to the traditional concurrency control setting. Specifically, our goal in this paper was two-fold:

- To illustrate via detailed examples that allowing transactions to access extra data not only improves the functionality of transactions but also has many performance benefits.

  We considered extra-data items that occur in typical database systems such as the log, the clock, and the concurrency control information. We also showed that other types of extra data, such as proclaimed values, and control data used for coordinating pipelined and cooperating transactions prove very useful for structuring transactions in order to improve performance.

- To investigate, in detail, the correctness issues that arise when transactions are allowed access to extra data.

  We saw that while traditional serializability can continue to be the mainstay of correctness, one needs to also consider extra-data independence of transactions. To precisely capture the interactions due to extra-data access, we axiomatized the operations on extra-data items to determine the serialization ordering requirements induced by extra-data access. These helped characterize the different types of correctness issues that must be considered. In this regard, we studied a variety of correctness notions.

Allowing transactions to access extra-data improves transaction functionality. On the other hand, as we discussed at several points in the paper, performance consequences can be either positive or negative depending on the properties of the data. One implication is that extra-data access must be allowed only if the consequences are not detrimental to performance, and if they are and yet extra data must be accessed, one must apply the least restrictive correctness criterion that fits the needs.

Some of the practical implications for the mechanisms used for transaction processing remain to be investigated. For instance, the transaction processing system must keep track of $\mathcal{C}$ relationships that are induced when transactions access not just the database but also the extra data. Also, it must ensure that operations are scheduled in such a way that acyclicity of $\mathcal{C}$ relations is ensured.

Acknowledgements

Our sincere thanks to Panos Chrysanthis, Lory Molesky, and Avi Silberschatz for their comments on previous versions of this paper.

References

[1] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proceedings of the seventeenth International Conference on Very Large Databases*, pages 103–112, 1991.

[3] D. Georgakopoulos, M. Rusinkiewicz and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the IEEE Seventh International Conference on Data Engineering*, 1991.

[4] N. Gehani and O. Shmueli. The LOG as Part of the Database. *Bell Laboratories Technical Memorandum*, 1992.

[5] J.N. Gray and A. Reuter, "Transaction Processing: Techniques and Concepts", Morgan-Kaufman, 1992.

[6] H.V. Jagadish and O. Shmueli. A Proclamation-Based Model for Cooperating Transactions. In *Proceedings of the eighteenth International Conference on Very Large Databases*, pages 265–276, 1992.

[7] Korth H. F. and G. Speegle. Formal Models of Correctness without Serializability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–386, 1988.

[8] K. Ramamritham and P. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties" in *Distributed Object Management*, Ozsu, Dayal, and Valduriez Ed., Morgan Kaufmann Publishers, 1992.

[9] K. Ramamritham Real-Time Databases. *International Journal of Distributed and Parallel Databases*, Vol. 1, No 2, 1993.

[10] L. Sha *Modular concurrency control and failure recovery— Consistency, Correctness and Optimality.* PhD thesis, Department of Computer and Electrical Engineering, Carnegie-Mellon University, 1985.

[11] M. R. Stonebraker, "Hypothetical Data Bases as Views", *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp 224-229, May 1981.